Mapping Features to Aspects: A Model-Based Generative Approach

Uirá Kulesza[1], Vander Alves[2,3], Alessandro Garcia[3]
Alberto Costa Neto[2], Elder Cirilo[1], Carlos J. P. de Lucena[1], Paulo Borba[2]

[1]PUC-Rio, Computer Science Department, Rio de Janeiro - Brazil
{uira, lucena, ecirilo}@inf.puc-rio.br

[2]Informatics Center, Federal University of Pernambuco, Recife - Brazil
{vra, acn, phmb}@cin.ufpe.br

[3]Lancaster University, Computing Department, Lancaster - United Kingdom
garciaa@comp.lancs.ac.uk

Abstract. Handling the various derivations of an aspect-oriented software family architecture can be a daunting and costly task if explicit support is not systematically provided throughout early and late development artifacts. Aspect-oriented software development (AOSD) has been recently explored by several research works as a technique that enables software product line customization. However, the application of AOSD has been limited to modularize specific crosscutting features encountered in the implementation of software product-line architectures or frameworks. Only a few works have investigated the development of product derivation approaches for AOSD. This paper presents a model-based generative approach to mapping features to aspects across different artifacts of an product line. Our main aim is to enable the smooth and systematic derivation of aspect-oriented software family architecture. Our approach is complementary to a set of previously-proposed modularization guidelines to implement aspect-oriented frameworks. We present details about the suite of mappings supported by our generative model, illustrate them in heterogeneous case studies, and discuss several implementation issues for its accomplishment.

1 Introduction

Aspect-Oriented Software Development (AOSD) [13, 19] is a software engineering approach to modularize crosscutting concerns that existing paradigms are not able to capture explicitly. Crosscutting concerns are broadly-scoped features or properties that often crosscut several modules in a software system. AOSD encourages modular descriptions of crosscutting concerns typically through a new modular unit, called *aspect*. Early aspects [5, 30] refer to the aspect-oriented (AO) approaches which address the explicit handling of crosscutting concerns at the requirements and architecture level. The majority of the existing development techniques, including the early aspects approaches, have explored the use of AO techniques to modularize traditional (such as logging and security) and domain-specific crosscutting concerns (e.g. [22]).

The use of AOSD techniques in the development of framework and product lines have been only recently exploited [1, 2, 4, 14, 22-28, 33]. However, aspects have been notouriously used to modularize crosscutting features encountered in the implementation of aspect-oriented software family architectures. We have identified from our experience [22-24] the following benefits on the application of AO techniques in product line

development: (i) clear separation and variation of crosscutting features starting at early phases; (ii) direct mapping of crosscutting features in aspects; (iii) simplied implementation of code generator, because the composition of crosscutting features is accomplished by aspect weavers; and (iv) improved reuse of artifacts associated with crosscutting features. However, the achievement of such benefits is fundamentally dependent on the provision of a set of guidelines to model, implement and compose non-crosscutting and crosscutting features. Also, it requires the definition of mapping rules between the different abstractions (such as, features, aspects, use cases) used in both the product line development and the product derivation stages.

This paper presents a model-based generative approach to mapping features from the problem space to aspects from the solution space. It is centered on the development of a generative model composed by three elements: (i) an architecture model; (ii) a feature model; and (iii) a configuration model. Our main aim is to support automatic product derivation of AO family architectures. Our approach is complementary to a set of previously proposed modularization guidelines [23, 24] to implement framework and product lines using aspect-oriented programming. We also present mapping rules between the kinds of features and implementation elements which guide the specification of our configuration model, and illustrate our approach with case studies.

This paper is organized as follows. Section 2 gives an overview of our AO approach for framework development. Section 3 presents our model-based generative approach by detailing the different models used in the product derivation stage. Section 4 illustrates the approach with a real-life case study in the mobile games domain. Section 5 discusses a set of lessons learned from the use of the proposed guidelines and mapping rules. Related work is discussed in Section 6. Section 7 offers concluding remarks.

2 On the Development of Aspect-Oriented Frameworks

Our previous work [23] has proposed an approach for developing AO application frameworks. This section briefly shows an overview of our framework development approach (Section 2.1) and the use of the approach in the JUnit framework (Section 2.2).

2.1 Approach Overview

In our approach, an OO framework specifies and implements not only its common and variable behavior using OO classes, but it also exposes a set of extension join points (EJPs) [23, 24] which can be used to also extend its functionality. EJPs establish a contract between the framework classes and a set of aspects extending the framework functionality. They aim at increasing the framework variability and integrability by serving two purposes: (i) to offer a set of join points spread and tangled in the framework classes into which the implementation of crosscutting optional and alternative features can be included; and (ii) to expose a set of framework events that can be used to notify or to facilitate a crosscutting integration with other software elements (such as, frameworks or components).

In this context, EJPs document crosscutting extension points for software developers that are going to instantiate and evolve the framework. They can also be viewed as a set of constraints imposed on the whole space of available join points in the framework design, thereby promoting safe extension and reuse. EJPs are a specialization of the Crosscutting Interface (XPI) concept [15, 31] applied in the context of

framework/product line development. A key characteristic of EJPs is that framework developers and users do not need to learn totally new abstractions to use them, as they can mostly be implemented using the mechanisms of AOP languages. We have presented guidelines to implement EJPs using AspectJ [24], including the codification of runtime and compile contracts which address the specification of constraints between the framework and its respective extension aspects.

Our approach promotes framework development as a composition of a core structure and a set of extensions. A framework extension can define one of the following: (i) the implementation of optional or alternative framework features; or (ii) the integration with an additional component or framework. The composition between the framework core and the framework extensions is accomplished by different types of extension aspects, each one defining a crosscutting composition with the framework by means of its exposed EJPs. We next describe the main concepts of our approach:

(i) *framework core* – implements the mandatory functionality of a software family. Similar to a traditional OO framework, this core structure contains the frozen-spots that represent the common features of the software family and hot-spot classes that represent non-crosscutting variabilities from the domain addressed;

(ii) *aspects in the core* – implement and modularize existing crosscutting concerns or roles in the framework core. They represent the traditional use of AOP to simplify the understanding and evolution of the framework core;

(ii) *variability aspects* – implement optional or alternative features existing in the framework core. These elements extend the framework EJPs with any additional crosscutting behavior;

(iii) *integration aspects* – define crosscutting compositions between the framework core and other existing extensions, such as an API or an OO framework. These elements also rely on the EJPs specification to define their implementation.

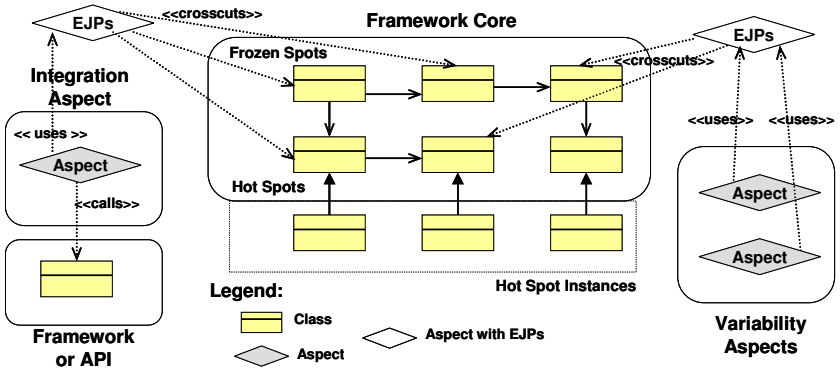

Fig. 1. Elements of our Framework Development Approach

Figure 1 shows the design of an OO framework with aspects following our approach. Both variability and integration aspects intercept only join points matched by pointcuts in the EJPs provided by the framework; further, such aspects must comply with all the constraints defined by the EJPs. This brings systematization to the framework extension and composition with other artifacts, providing a number of benefits [23], such

as enhanced understandability and evolution of the framework core, better management of features, safe framework reuse, and pluggable/unpluggable crosscutting framework extensions.

2.2 JUnit Framework: An Ilustrative Example

The main purpose of the JUnit framework is to allow the design, implementation and execution of the unit tests in Java applications. Figure 2 presents the main elements of the JUnit architecture. Its main functionalities are: the definition of test cases or suites to be executed (`TestCase` and `TestSuite` subclasses); the execution of a selected test case or suite (`BaseTestRunner` and `TestRunner` classes); and the collection (`TestResult` class) and visual presentation of the test results. However, different extensions can be implemented to add new functionalities into the JUnit framework core. Some examples of simple extensions are the following: (i) enable JUnit to execute each test suite in a separate thread (`ActiveTestSuite` aspect), and wait until all tests finish. In order to implement this extension we need to observe the event when the test suite starts running, the event when each test method runs, and the event when the test suite stops running; (ii) enable JUnit to run each test repeatedly (`RepeatedTestGeneric` aspect). In order to implement this extension we need to observe the event when each test method runs.

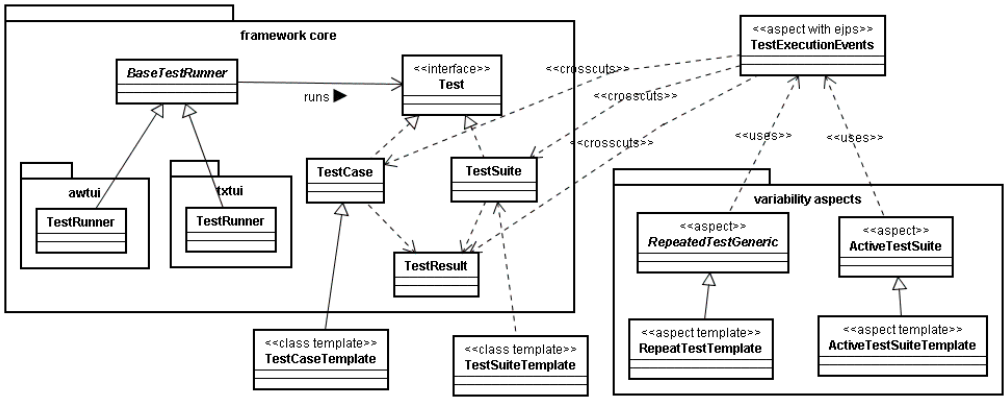

Fig. 2. JUnit Aspect-Oriented Architecture

These JUnit extensions need to observe internal events, which are spread over JUnit classes. In other words, such extensions are not well modularized in the object-oriented design. In our approach, an EJP was used to expose such key events that are not adequately captured by the OO design and that are useful for crosscutting compositions scenarios. Figure 2 presents an EJP, called `TestExecutionEvents`, which exposes a set of join points in the JUnit framework. Some of these join points were discovered by checking them against these anticipated crosscutting extension scenarios. Based on this first set of discovered join points, we could foresee other relevant events that may be of interest when extending JUnit.

3 Mapping Features to Aspect-Oriented Architectures

This section presents the model-based generative approach to mapping features to aspect-oriented implementation elements. It enables deriving members of an AO software family architecture through model-based generative support for customizing architecture variabilities. Our approach is presented in a stepwise fashion. First, the three main models of our generative approach are described by showing how the domain engineering team can develop and derive them during or after the architecture implementation (Sections 3.1-3.3). Later subsections detail the activities of template implementation (Section 3.4) and product derivation (Section 3.5).

3.1 Specifying an Architecture Model

The implementation of a software family architecture typically results in a set of implementation elements/artifacts, such as, classes, interfaces, templates, aspects, and extra files (e.g. configuration and image files). An aggregation of many of these heterogeneous elements realizes the components previously defined in design of the AO family architecture. For example, our aspect-oriented approach to framework implementation (Section 2) generates a set of well-defined implementation elements, such as: the framework core classes, OO hot-spots classes and interfaces, aspects in the core, extension join point aspects, variability aspects, and integration aspects.

The first step of our approach consists of the specification of an architecture model. It associates the implementation elements from the software family with the specification of its architectural components. The main purpose of the architecture model is to create a representation of these implementation elements in order to relate them to feature models, where the component variabilities are expressed. It is developed to be used and processed by our model-based generative tool. An architecture model is composed by a set of components. Each component can aggregate different implementation elements, such as, classes, interfaces, aspects, templates and extra files. Templates are used to codify implementation elements (classes, interfaces, aspects and configuration files) which need to be customized during the software family derivation. A component can also be composed of a set of sub-components.

An architecture model can also be automatically generated by traversing the directory that maintains the implementation elements. Specialized AST (Abstract Syntax Tree) APIs available for many programming languages can be used to help the implementation of this traversing function. The functionality of reverse engineering from code to models (e.g. Java code to UML class diagrams) implemented by many IDEs (Integrated Development Environments) has a similar purpose to this functionality. Templates are the only implementation elements that need to be codified to specify the architecture model. They can be used to specify: (i) subclasses/subtypes of framework hot-spots (classes or interfaces); (ii) concrete subaspects that implement any implementation alternative for a crosscutting feature; or (iii) any class, aspect or configuration file that needs to be customized during the product derivation process.

Figure 3(c) shows an example of an architecture model implemented using the EMF (Eclipse Modeling Framework) [6] technology. It partially represents the JUnit aspect-oriented architecture described in Section 2.2. It is composed of two main components: core and extensions. Each of them aggregates the implementation elements

respectively for: (i) the framework core, including the graphical user interfaces available, and (ii) the extension aspects. The `TestExecutionEvents` aspect implements the JUnit EJP and is part of the core component. The following templates were also specified for the JUnit architecture model: (i) `TestSuiteTemplate` and `TestCaseTemplate` which are used to further derivation of specific test suite and test case classes; and (ii) `ActiveTestSuiteTemplate` and `RepeatedTestTemplate` which will be used to derive variability aspects that respectively address the concurrent and repeated execution of tests to be applied to test suites and test cases, respectively.

3.2 Specifying the Feature Model

The following step in the definition of the generative artifacts is to specify the feature model for the AO software family architecture. A feature model [7] is used to represent the common and variable features of a software family. Since the pivotal purpose of the generative models is to automate product derivation, the feature modeling focuses mainly on the specification of the software architecture variabilities. In our approach, we use the feature models proposed by Czarnecki et al [8], which allow modeling mandatory, optional, and alternative features, and their respective cardinality. The feature modeling plugin (FMP) [3] supports the modeling of feature models in Eclipse IDE. Figure 3(a) presents the feature model of the JUnit framework. It defines three main features: (i) Testing – which defines the test suite and cases specific of an application; (ii) Runner – represents the graphical user interfaces alternatives provided by JUnit; and finally (iii) Extensions – which defines fine-grained extensions to be applied to test suites and cases.

Our approach defines a simple extension [26, 25] to the feature model in order to enable the aspect customization during product derivation. Our extension defines two properties, called `<<crosscutting>>` and `<<joinpoint>>`, which can be assigned to specific features being modeled. A *crosscutting feature* is used to represent aspects from the architecture model that can extend the behavior of other system features. A *joinpoint feature* is used to represent specific join points from implementation elements of the software family architecture. These join points are candidates to be extended by aspects in the solution space. Figure 3(a) shows examples of such properties in the JUnit feature model: (i) each one of the extension features are modeled as crosscutting; and (ii) the test suite and test cases features are modeled as joinpoint features.

The crosscutting and join point features are mapped, respectively, to the following implementation elements: extension aspects and extension join points (EJPs). In order to enable the customization of extension aspects to affect only specific EJPs, crosscutting relationships between crosscutting and join point features can be specified during product derivation [26]. Details about the mapping of these features to implementation elements will be presented in next sections.

3.3 Specifying the Configuration Model

Our approach defines the mapping between features and implementation elements by means of a configuration model. The configuration model is defined based on a set of mapping rules. It expresses the configuration knowledge existing in generative software development [7]. The specification of configuration models makes it possible to reason about configuration knowledge separatedly from the problem space (feature model) and

the solution space (architecture model). It also allows smoothly realizing various changes in the configuration knowledge, such as discarding or modifying the existing mapping.

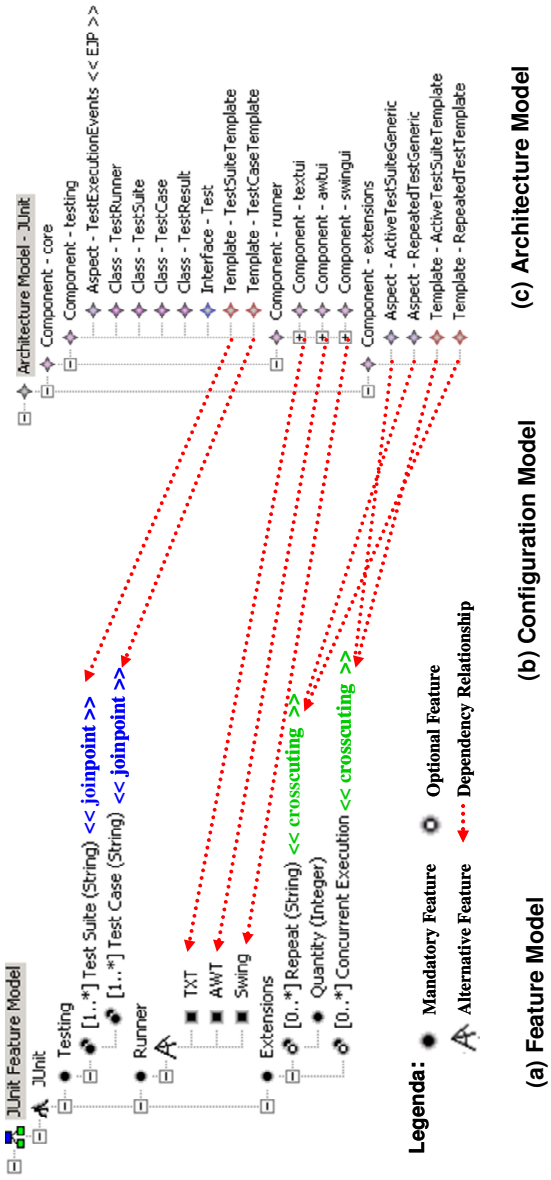

Fig. 3. JUnit Feature, Configuration and Architecture Models

Our configuration model is composed of three elements: (i) dependency relationships between implementation elements existing in architecture models and

features in feature model specifications; (ii) definition of valid crosscutting relationships between join point and crosscutting features; and (iii) specification of the mapping between join point features and concrete join points existing in implementation elements from the architecture model. Table 1 presents the elements of the configuration model and their respective purpose for product derivation. In the following, we describe the guidelines to specify such configuration elements.

Table 1. Configuration Model Elements

Configuration Model Element	Main Purpose
Dependency Relationships between Implementation Elements and Features	• Choice of Variabilities
Valid Crosscutting Relationships between Crosscutting and Joinpoint Features	• Restriction of Crosscutting Relationships in the Problem Space
Mapping between Joinpoint Features and Concrete Joinpoint from EJPs	• Customization of Aspects Pointcuts

The dependency relationships between implementation elements (components, classes, aspects, templates and files) and features are used to determine which implementation elements must be instantiated when a specific set of features are selected in product derivation. They represent the accomplishment of decision models [32]. Table 2 describes the mapping rules between: (i) the kinds of existing features; and (ii) the elements from our AO approach to framework implementation (Section 2). These mapping rules can be used as a base to derive the dependency relationships between implementation elements and features.

As we can see in Table 2, mandatory features are mapped to mandatory implementation elements (such as the framework core and the aspects in the core). In the configuration model, however, there is no need to create any dependency relationship between these elements, since these mandatory implementation elements must be found in every member/product of the software family/product line. As we mentioned in Section 3.2, the feature models are used to support the product derivation. It implies that they are mainly adopted to represent the software family variabilities. The representation of mandatory features is optional. However, if it is of interest to explicitly maintain the mapping between all the features to implementation elements, additional dependency relatioships can be also created in the configuration model. Table 2 also shows that: (i) both optional and crosscutting alternative features are mapped to variability or integration aspects; and (ii) join point features are mapped to EJP implementation elements.

Besides using the mapping rules to define the dependency relationships, the following guidelines can be used to specify them: (i) if an implementation element must be instantiated to every member of a software family, there is no need to create a dependency relationship to any feature; and (ii) if an implementation element depends on

any feature occurrency, a dependency relationship must be created between them. In the case of the template implementation element, the dependency relationships define if they must be processed to generate any specific element to be included in the product/member generated. Thus, each template depends necessarily of a feature, which provides the useful information for the template processing.

Table 2. Mapping Rules between Features and Implementation Elements

Feature Type	Implementation Element
Mandatory Features	• Framework Core • Aspects in the Core
Alternative Features	• Hot-Spots Classes in the Framework Core
Joinpoint Features	• Extension Join Points (aspects)
Optional Features	• Variability and Integration Aspects
Alternative Crosscutting Features	• Variability and Integration Aspects

Figure 3(b) defines a set of dependency relationships for the JUnit configuration model. The GUI alternative components (textui, awtui, swingui) and their respective implementation elements (classes, interfaces, etc) will be instantiated based on the alternative feature selected for the Runner feature. As we can see, every template from the JUnit architecture model depends necessarily on a specific feature. The `RepeatedTestTemplate`, for example, will be processed only if the application engineering requests a Repeat feature during product derivation stage.

Our configuration model also defines a set of valid crosscutting relationships that can occur between crosscutting and join point features. These relationships are used to restrict which aspects (represented as crosscutting features) can affect which join points from the classes/aspects of the software family (represented as join point features). In our approach, a code generator uses such information to verify if application engineers are specifying valid relationships between crosscutting and join point features. Table 3 shows the valid crosscutting relationships for the JUnit case study. It specifies that: (i) the Repeat crosscutting feature can only extend the behavior of Test Case features; and (ii) the Concurrent Execution feature can only extend Test Suite features.

The last component of our configuration model is the mapping between the join point features and concrete join points existing on the implementation elements from the architecture model. This mapping is used by a code generator to customize pointcuts from extension aspects. If all the extension aspects have fixed pointcuts, there is no need to specify this mapping. The concrete join points can be directly found and extracted from the EJPs defined for the software family architecture. Table 3 shows the joinpoint

mapping of the TestCase and TestSuite features to concrete join points exposed by the EJPs of the JUnit framework.

Table 3. Additional Elements of the JUnit Configuration Model

Configuration Model Element	JUnit Framework
Valid Crosscutting Relationships	Repeat feature <<crosscuts>> Test Case feature Concurrent Execution <<crosscuts>> Test Suite feature
Mapping between Joinpoint Features and Joinpoint from EJPs	Test Case feature <<maps>> TestExecutionEventsEJP. testCaseExecution(…); Test Suite feature <<maps>> TestExecutionEventsEJP.testSuiteExecution(…);

3.4 Template Implementation

The last activity of product-line/domain engineering is to codify the templates for the architecture model (Section 3.1). During the specification of the architecture model, templates are defined to specify implementation elements which define any variability on their structure. After that, dependency relationships between templates and features are created in the configuration model specification (Section 3.3). The implementation of templates depends on the information provided by the feature model. The feature model is used to collect any data that helps to customize the template variabilities. For this reason, the complete codification of templates can only be completely realized after the specification of the architecture, feature and configuration models.

There are many tools which implement the template technology [9]. In our particular tool implementation, we are using JET (Java Emitter Templates) to codify our templates. JET is the template engine of the Eclipse Modeling Framework (EMF) plugin. It can be used to implement templates for any kind of implementation element (classes, aspects, configuration files). Figure 4 shows the implementation of the `TestSuite` template using JET. It contains initially basic configuration code of a JET template (lines 1-5). The `FeatureElement` type is used to store a reference to the feature which the template depends on the configuration model. This variable is suitably configured by the code generator during product derivation.

For example, the `TestSuite` template depends on the test suite feature. Because of that, the code generator will process this template for each test suite feature specified; it will also use the information from a specific test suite feature specified during this processing. The processing of the template `TestSuite` causes the code customization of test suite classes by using information from the `FeatureElement` attribute type, such as: (i) the class name of the test suite (line 9); and (ii) which test case classes will be part of this test suite (lines 13-18). Templates of aspects can also use the information about the

join point mapping from the configuration model to customize pointcuts. This information is made available in the `FeatureElement` class[1] through a specific method (`getJoinPointFeatures()`).

```
01 <%@ jet package="translated"
02    imports="org.eclipse.emf.common.util.Elist ..."
03    class="TestSuiteTemplate" %>
04
05 <% FeatureElement testSuite = (FeatureElement) argument;%>
06 import junit.framework.Test;
07 import junit.framework.TestSuite;
08
09 public class <%=testSuite.getName()%>TestSuite {
10
11    public static Test suite(){
12       TestSuite suite = new TestSuite("<%=feature.getName()%>");
13      <% EList features = feature.getChildren();
14         for (Iterator iter=features.iterator(); iter.hasNext();){
15           FeatureElement testCase= (FeatureElement) iter.next(); %>
16        suite.addTest(
17                 new TestSuite(<%=testCase.getName()%>Test.class));
18      <% } %>
19      return suite;
20    }
21
22 }
```

Fig. 4. The `TestSuite` template

3.5 Product Derivation

In the product derivation phase, an instance of the software family architecture is created based on the choice of variabilities by the software developers. It is supported in our approach by two main activities: (i) choice of variabilities through a feature model instance; and (ii) choice of valid crosscutting relationships between features. In order to generate a member of the software family architecture, a code generator uses the information collected by the derivation activities in addition to the architecture and configuration models.

The product derivation phase is composed of three steps: (i) initially a software developer specifies a feature model instance and its respective crosscutting relationships; (ii) next, a code generator uses this feature model instance and a configuration model, to decide which elements from the architecture model must be part of the instance generated; and (iii) finally, after processing and customizing the architecture model, the code generator loads the implementation elements that will constitute the final product in a specific folder or project created in an IDE, such as Eclipse.

The following actions must be performed by the code generator during the product derivation process:

---

[1] The `FeatureElement` class is used in our plugin to store the subtree of a specific feature from a feature model configuration created using the FMP plugin.

(i) **verification of valid crosscutting relationships** – the code generator must initially guarantee that only valid crosscutting relationships were created by the software developers. The detection of invalid crosscutting relationships would interrupt the product derivation process in order to avoid subsequent problems in code generation;

(ii) **processing of the architecture model** – the code generator processes the architecture model by traversing all the implementation elements. It proceeds as follows. For each component encountered the code generator verifies in the configuration model if it depends on any special feature. In this case, the code generator only instantiates[2] that component (and processes its respective sub-elements) if there is an occurrence of that feature in the feature model instance. When processing implementation elements from each component the same process is applied, that is, it is verified if the implementation element depends on specific features as a condition to instantiate it. As we mentioned before, template elements always depend on some feature. They are processed by the code generator for each occurrence of that feature. During a template processing, all the information about the feature and respective sub-features, which it depends, is used to support the template customization (Section 3.4);

(iii) **customization of aspect pointcuts** – during the processing of aspect templates, pointcuts can also be customized. Every aspect template must depend on a specific crosscutting feature. If the aspect template has any pointcut to be customized, its join points can be obtained by looking at which join point features are affected by the crosscutting feature representing the aspect template. This information is obtained by the code generator in the configuration model and it is used by the template to customize its respective variable pointcuts.

4 J2ME Games Product Line

J2ME games are mainstream mobile applications of considerable complexity [1]. In this case study, we implemented the generative model of an industrial J2ME game Software Product Line (SPL) based on EJPs (Section 2). The case study implementation exposed game core EJPs in order to allow the composition of crosscutting extensions in its basic functionality. The detailed architecture model is described elsewhere [24]. The resulting generative model for this SPL is shown in Figure 5.

According to Figure 5, there are EJPs in the core (`ResourceEvents` and `DrawingEvents`) as well as in specific extension components (`BrightEvents`, `CloudEvents`, and `ScreenEvents` in Components Bright, Clouds, and Image Loading, respectively). EJPs in the core are due to the mandatory features FLIP and ImageLoading, and are not necessarily linked to such features since they are in every SPL instance. On the other hand, the EJPs in extension components are due to specific alternative subfeatures of those features or the optional features Bright and Clouds, and need to be linked to such features since they are present in only some SPL instances.

Additionally, the dependencies for the Bright component have a peculiar behaviour: when the Bright feature is *not* selected, then the NoBright aspect has to be

[2] In Java language, for example, the component instantiation can be mapped to the creation of a package, which aggregates the implementation elements from that component.

included. When such a feature is selected, the Bright Aspect is included, instead. This occurs because the variability related to this feature could not be extracted into a single module (an aspect) to be composed with a base module representing non-brightness behavior. Therefore, the generative model for this feature has limited compositionality, which may lead to reduced scalability. Althought not desirable, this phenomenon can arise frequently in SPLs. We further discuss this in Section 5.1.

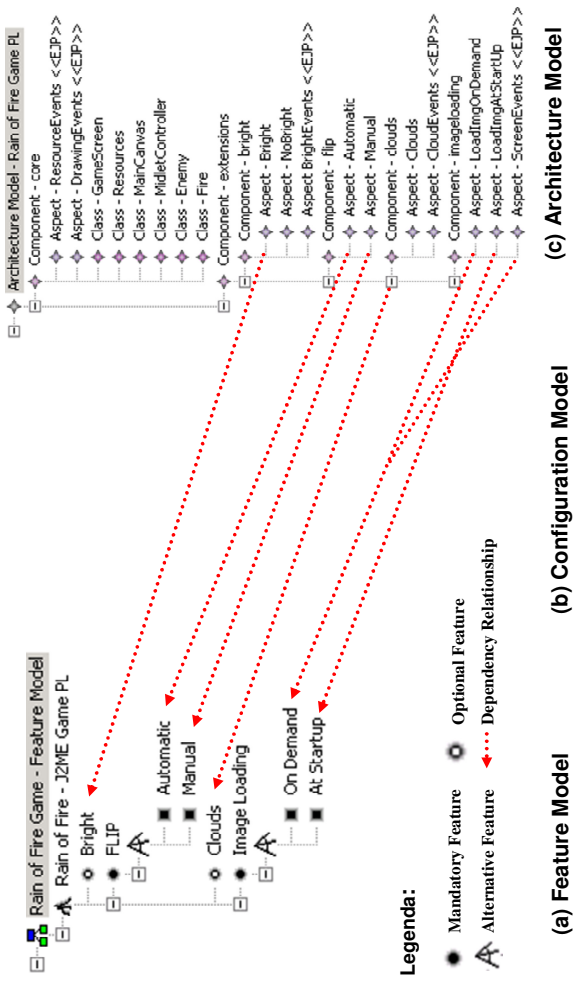

Fig. 5. J2ME Game Product Line Configuration and Architecture Models

In terms of granularity, the dependency relationships can depart from either fine-grained implementation elements such as aspects and EJPs, or from coarse-grained implementation elements such as components (Clouds and Bright). This reflects inherent levels of granularity in the SPL variability. Based on the lessons learned from new case studies, we intend to derive guidelines which help to reduce the total amount of

dependency relationships, and consequently, facilitate the maintenance of the configuration model.

In this case study, we did not have occurrences of crosscutting relationships between crosscutting and join point features. The reason for this is that such relationships occur more frequently in the context of homogenous variability aspects, whereas in this case study most variability aspects were heterogeneous and with fixed pointcuts. For the same reason, this case study did not customize aspect pointcuts, thereby not requiring mapping between joinpoint features and concrete joinpoint from EJPs.

5 Discussion and Lessons Learned

This section provides discussion and some lessons learned based on our experience on applying our proposed approach. In particular, it shows the benefits that our implementation guidelines and mapping rules can bring: (i) to deal with the problem of feature interaction (Section 5.1); to specify traceability links between features, use cases and extension join points (Section 5.2) and; to adopt proactive, reactive or extractive product line development strategies (Section 5.3).

5.1 Feature Interaction

The development of frameworks and product lines involves the modularization and composition of different features. The interdependence and interference between features can generate the problem of feature interactions. A feature interaction [36] occurs when a feature is modified or influenced by the behavior of another or other features. When features interfere with each other and are implemented by aspects, we say there is an aspect interaction. Different types of aspect interactions can occur, Sanen et al [37] proposes a classification of four types: (i) mutual exclusion - two or more aspects cannot coexist in the same application; (ii) dependency - an aspect depends on another one to execute correctly; (iii) reinforcement - there is a positive influence from one aspect to another one; and (iv) conflict - an aspect works correctly in isolation, but not in the presence of other aspects.

Different approaches can be used to deal with the problem of feature/aspect interaction during product line development. The mutual exclusion and dependency interactions, can be addressed, respectively, by the definition of <<excludes>> and <<requires>> constraints in a feature model [7]. On the other hand, conflicts and reiforcement between aspects need to be designed and implemented carefully. Conflicts in the order of composition of aspects, for example, can require the definition of precedence statements (available in AspectJ). Our approach contributes in two ways to deal with feature interaction situations: (i) the specification of EJPs helps to identify possible join points of interactions between aspects from frameworks and product lines; and (ii) the explicit definition of valid crosscutting relationships between crosscutting and join point features in our configuration model allows to restrict undesirable aspect interactions.

Another potential problem of feature interaction is that the implementation element to which a certain feature is mapped may depend on the selection of another feature: if such other feature is also selected, then the implementation element to which the former feature is mapped might be different. In such case, there is reduced

compositionality of the configuration model because functional composition at the feature model does not reflect into functional composition at the architectural model. Therefore, this may comprise the scalability of the resulting SPL. However, we note that compositionality, despite desirable, is not always possible at the architectural level. Indeed, the very existence of EJPs indicates that interactions among the SPL core and its extensions constrain arbitrary composition.

5.2 Use Case Extensions

Use cases [17] is a largely adopted technique for requirements specification. It has been adopted by many modern development software processes. A use case is defined as a sequence of actions performed by the system to provide an observable result of value to a particular user. The use cases technique also provides an extension mechanism which can be used to describe extra (mandatory or optional) behavior. The extend relationship can be defined between use cases to achieve such extension purpose. An extension use case can add behavior in specific set of extension points of other use cases. Jacobson [16] argues that the use case extension mechanism can be used to model aspects during the requirements specification. In this author's approach, extension use cases model aspects and the extension points represent join points at the requirements level.

This paper presented a set of mapping rules from features to implementation elements of our aspect-oriented framework development approach. We have also noticed that there is a strong synergy between our extension join points (EJPs) and the use case extension points proposed by Jacobson. The use cases extension points are natural candidates to be implemented as an EJP. Alike, extension use cases can be implemented as extension aspects. The existence of mapping rules between feature models, use cases and implementation elements from our approach can help us to define and keep traceability links [18] between the different artifacts developed in the requirements, architecture and implementation stages. It can improve the development or evolution of frameworks and product lines by supporting relevant software engineering activities, such as change impact analysis and consistent variability management. We are going to develop new case studies to explore the synergies between these techniques and assess how their integration can be used to better support the management of traceability links between artifacts. These traceability links would also be specified as a set of mappings between the mentioned and other modeling notations in order to support round-tripping and, as a consequence, to address multi-level customization approaches [11] with the use of aspect-oriented abstractions, such as, EJPs and crosscutting/join point features.

5.3 Software Product Line Adoption Approaches

Different adoption strategies [21] can be used to develop software product lines (SPLs). The proactive approach motivates the development of product lines considering all the products in the foreseeable horizon. A complete set of artifacts to address the product line is developed from scratch. In the extractive approach, a SPL is developed starting from existing software systems. Common and variable features are extracted from these systems to derive an initial version of the SPL. The reactive approach advocates the incremental development of SPLs. Initially, the SPL artifacts address only a few products. When there is a demand to incorporate new requirements or products, the common and variable artifacts are incrementally extended in reaction to them.

Our approach has been used mainly in the aspect-oriented refactoring [29] of existing object-oriented frameworks and product lines [23, 24]. However, we believe that our guidelines to implement crosscutting features using aspects can be used in different SPL adoption strategies. Also the mapping rules presented in this paper play an important role in the adoption strategies since they show clearly the relationships between features and implementation elements. In the extractive approach, our guidelines and mapping rules can be useful to determine which optional, integration and alternative features would be implemented using aspects. The reactive approach can benefit from EJPs previously specified to introduce new demands of optional and integration crosscutting features in the SPL or framework core. EJPs also promote a weak coupling between a SPL core and its respective crosscutting extensions. This can help in the incremental development or evolution of SPLs by allowing to (un)plug existing features and simplifying the core complexity. Finally, the proactive approach can benefit specially from the mapping between use case extensions, features and extension aspects presented previously (Section 5.2). Existing product-line development processes can use these mapping guidelines to help in the analysis and identification of candidate aspects to implement specific features.

6 Related Work

There are many feature-based tools to product derivation available in the industry, such as Pure::variants and Gears. Pure::variants [34] is a SPL tool for feature modelling and product derivation. It allows specifying a generative model, where features are modelled graphically in different formats such as trees and tables and constraints among features are expressed using first order logic in Prolog. Architecture modelling is also possible and conforms to a specific meta-model. The configuration model is specified in terms of rules relating elements of both models and can make use of constumizable transformation engine. Similarly, Gears[35] allows the definition of a generative model focused on product derivation. However, its language for expressing constraints at feature models is propositional logic instead of full first-order logic. Additionally, Gears allows the definition of modular feature models, which can be combined hierarchically and support product line populations.

Framed Aspects approach [27] explores the instantiation of AO architectures. It proposes the integration between Frame and AOP technologies. The main difference between our approach and the Framed Aspects, is that they define many of the decision steps about the product derivation process in the template code of frames by means of meta-tags. In our approach, the decisions related to the architecture customization process are described separately by our configuration model. It makes easier to adapt or evolve the decisions related to the architecture customization. We also use feature model instances to gather all information necessary for the resolution of AO variabilities.

Griss [14] presents some benefits of integrated use of aspect-oriented implementation technologies and feature engineering during development of product lines. He also outlines a general methodology for the combination of these two technologies. Our work can be seen as a concrete method of development and derivation of framework and product lines using aspect-oriented techniques, which follows the general guidelines presented by Griss.

There are also similarities between the mappings from our model-based generative approach and the proposed in the feature-based model templates approach [10]. The dependency relationship from our configuration model can be seen as a special kind of the "presence condition" annotation proposed by those authors. Our central idea, however, is to keep these relationships and other configuration knowledge information in a complete and separate model.

7 Conclusion

This paper presented a model-based generative approach to mapping features to aspects in order to enable the derivation of AO software family architecture. The approach defines a generative model, encompassing an extension for feature modeling to represent crosscutting relationship among features, mapping rules and guidelines for product derivation, and an architecture model complementary to a set of previously-proposed modularization guidelines to implement aspect-oriented frameworks. We illustrated the approach with case studies and discussed its benefits and drawbacks for handling feature interaction and SPL adoption strategies. As future work, we plan to explore the synergies between use case extensions, features and extension aspects in the context of supporting the management of traceability links between artifacts.

References

[1] V. Alves, P. Matos, L. Cole, P. Borba, G. Ramalho. “Extracting and Evolving Mobile Games Product Lines”. Proceedings of SPLC'05, LNCS 3714, pp. 70-81, September 2005.

[2] M. Anastasopoulos, D. Muthig. “An Evaluation of Aspect-Oriented Programming as a Product Line Implementation Technology”. In Proceedings of the International Conference on Software Reuse (ICSR), pp. 141-156, July 2004.

[3] M. Antkiewicz and K. Czarnecki. FeaturePlugin: “Feature modeling plug-in for Eclipse”, OOPSLA’04 Eclipse Technology eXchange (ETX) Workshop, 2004.

[4] S. Apel, D. Batory. “When to Use Features and Aspects? A Case Study”. In Proceedings of GPCE'06, pp. 59-68, Portland, Oregon, October 2006.

[5] E. Baniassad, et al. “Discovering Early Aspects”. IEEE Software 23(1): 61-70, January 2006.

[6] F. Budinsky, et al. Eclipse Modeling Framework. Addison-Wesley, 2004.

[7] K. Czarnecki, U. Eisenecker. Generative Programming: Methods, Tools, and Applications, Addison-Wesley, 2000.

[8] K. Czarnecki, S. Helsen, U. Eisenecker. Staged Configuration Using Feature Models. In Proceedings of the Third Software Product-Line Conference, September 2004.

[9] K. Czarnecki, S. Helsen. Feature-Based Survey of Model Transformation Approaches. IBM Systems Journal, 45(3), 2006, pp. 621-64.

[10] K. Czarnecki, M. Antkiewicz. “Mapping Features to Models: A Template Approach Based on Superimposed Variants”. In Proceedings of GPCE'05, pp. 422-437, October 2005.

[11] K. Czarnecki, M. Antkiewicz, C. Kim. “Multi-level Customization in Application Engineering”. Communications of the ACM, Vol. 49, No. 12, pp. 61-65, December 2006.

[12] M. Fayad, D. Schmidt, R. Johnson. Building Application Frameworks: Object-Oriented Foundations of Framework Design. John Wiley & Sons, September 1999.

[13] R. Filman, T. Elrad, S. Clarke, M. Aksit. Aspect-Oriented Software Development. Addison-Wesley, 2005.

[14] M. Griss. “Implementing Product-Line Features With Component Reuse”. Proceedings of the International Conference on Software Reuse (ICSR’2000), Vienna, Austria, June 2000.

[15] W. Griswold, et al, "Modular Software Design with Crosscutting Interfaces", IEEE Software, Special Issue on Aspect-Oriented Programming, January 2006.

[16] I. Jacobson. “Use Cases and Aspects-Working Seamlessly Together”, Journal of Object Technology 2(4): 7-28 (2003).

[17] I. Jacobson, M. Christerson, P. Jonsson, G. Overgaard Object-Oriented Software Engineering: A Use Case Driven Approach, Addison-Wesley, 1992.

[18] M. Jarke, “Requirements Traceability,” Comm. ACM, vol. 41, no. 12,, pp. 32-36, Dec. 1998.

[19] G. Kiczales, et al. Aspect-Oriented Programming. Proc. of ECOOP’97, Finland, 1997.

[20] G. Kiczales, et al, "Getting Started with AspectJ," Comm. ACM, vol. 44, pp. 59-65, 2001.

[21] C. Krueger. “Easing the Transition to Software Mass Customization”. In Proceedings of the 4th International Workshop on Software Product-Family Engineering, pp. 282–293, 2001.

[22] U. Kulesza, A. Garcia, C. Lucena, P. Alencar. “A Generative Approach for Multi-Agent System Development”. In "Software Engineering for Multi-Agent Systems III". LNCS 3390, pp. 52-69, 2004.

[23] U. Kulesza, V. Alves, A. Garcia, C. Lucena, P. Borba. “Improving Extensibility of Object-Oriented Frameworks with Aspect-Oriented Programming”. In Proceedings of the 9th International Conference on Software Reuse (ICSR-9), pp. 231-245, June 2006.

[24] U. Kulesza, et al. “Implementing Framework Crosscutting Extensions with EJPs and AspectJ”, Proceedings of the ACM SIGSoft 20th Brazilian Symposium on SoftwareEngineering (SBES’2006), pp. 177-192, Florianópolis, Brazil, October 2006.

[25] U. Kulesza, A. Garcia, F. Bleasby, C. Lucena. “Instantiating and Customizing Product Line Architectures using Aspects and Crosscutting Feature Models”. Proceedings of the Workshop on Early Aspects, OOPSLA’2005, San Diego,2005.

[26] U. Kulesza, C. Lucena, P. Alencar, A. Garcia. “Customizing Aspect-Oriented Variabilites using Generative Techniques”. Proceedings of SEKE'06, pp. 17-22, San Francisco, 2006.

[27] N. Loughran, A. Rashid. “Framed Aspects: Supporting Variability and Configurability for AOP”. Proceedings of ICSR’2004, pp. 127-140, 2004.

[28] M. Mezini, K. Ostermann: “Variability Management with Feature-Oriented Programming and Aspects”. Proceedings of FSE’2004, pp.127-136, 2004.

[29] M. Monteiro, J. Fernandes. “Towards a Catalog of Aspect-Oriented Refactorings”. In Proceedings of AOSD ’05, pp. 111–122, Chicago, March 2005.

[30] A. Rashid, A. Moreira, J. Araújo. “Modularisation and Composition of Aspectual Requirements”. Proceedings of AOSD’2003, pp. 11-20, Boston, March 2003.

[31] K. Sullivan, et al. Information Hiding Interfaces for Aspect-Oriented Design, Proceedings of ESEC/FSE´2005, pp.166-175, Lisbon, Portugal, September 5-9, 2005.

[32] D. Weiss, C. Lai. Software Product-Line Engineering: A Family-Based Software Development Process, Addison-Wesley Professional, 1999.

[33] C. Zhang, H. Jacobsen. “Resolving Feature Convolution in Middleware Systems”. Proceedings of OOPSLA’2004, pp.188-205, October 24-28, 2004, Vancouver, BC, Canada.

[34] Pure::Variants, URL: http://www.pure-systems.com/, January 2007.

[35] Gears/BigLever, URL: http://www.biglever.com/, January 2007.

[36] M. Jackson, P. Zave. “Distributed feature composition: A virtual architecture for telecommunications services”. IEEE Transactions on Software Engineering Vol.24,No.10,pp.831-847, October 1998.

[37] F. Sanen, et al. “Classifying And Documenting Aspect Interactions”. Proceedings of the 5th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS) at AOSD 2006, Bonn, Germany, March 2006.